编译器

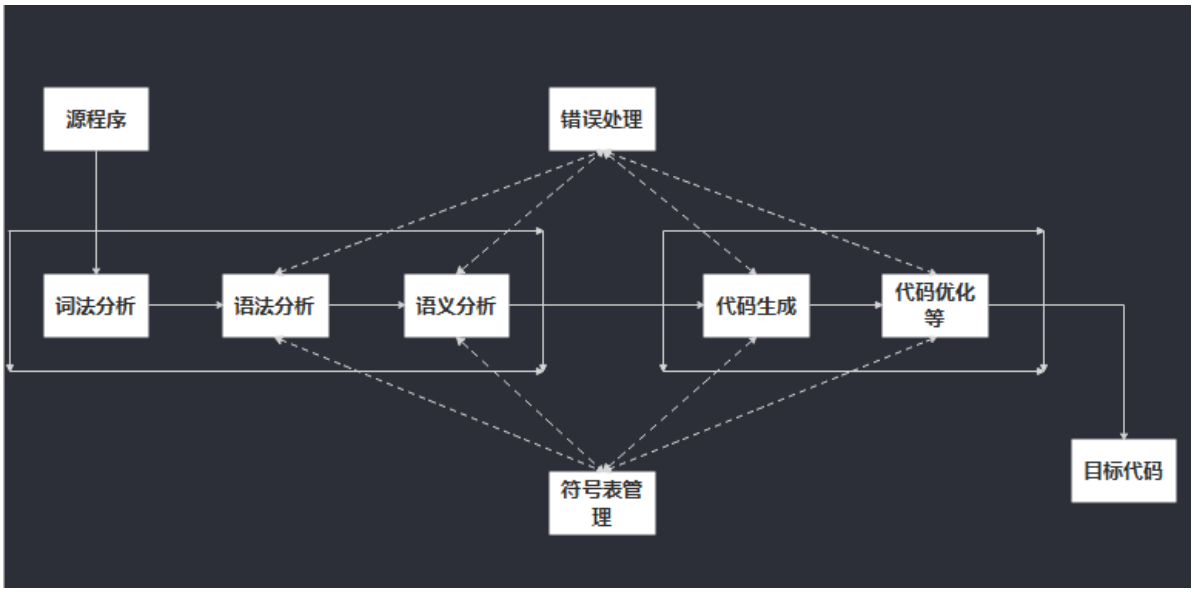
为一个具体语言编写一个编译器是一个较为复杂的任务，需要耗费较多的时间，查阅学习各种书籍，整体完成后的代码量应该是不容小觑的。仅仅是完成一个简单的词法分析就需要400行代码完成，且还不是最终版本。

编译器构造

下图给出编译器的构造方式，分成两个部分。

分析部分（前端）： 由词法分析、语法分析、语义分析所构成的分析部分的作用是对源程序文本进行分析，最后生成中间代码。词法分析将源程序的文法分析出文法类别码提供给语法分析程序，语法分析将类别码进一步组合成语法成分信息，如。语义分析对语法成分进行进一步的分析，识别出具体含义并且生成中间代码。

综合部分（后端）： 通过代码生成和代码优化生成目标程序。代码生成部分将源程序的中间形式转换为汇编语言或者机器语言。进行代码优化的主要目的是要获得更高效的目标程序，在确保源代码功能不变的前提下，使目标代码更简短，减少存储空间和运行时间。



分析过程中重要的定义

1. 类别码

单词名称	类别码	单词名称	类别码	单词名称	类别码	单词名称	类别码
Ident	IDENFR	!	NOT	*	MULT	=	ASSIGN
IntConst	INTCON	&&	AND	/	DIV	;	SEMICN
FormatString	STRCON	||	OR	%	MOD	,	COMMA
main	MAINTK	while	WHILETK	<	LSS	(	LPARENT
const	CONSTTK	getint	GETINTTK	<=	LEQ	)	RPARENT
int	INTTK	printf	PRINTFTK	>	GRE	[	LBRACK
break	BREAKTK	return	RETURNTK	>=	GEQ	]	RBRACK
continue	CONTINUETK	+	PLUS	==	EQL	{	LBRACE
if	IFTK	-	MINU	!=	NEQ	}	RBRACE
else	ELSETK						

2. 文法

```
编译单元  CompUnit → {Decl} {FuncDef} MainFuncDef  // 1.是否存在Decl 2.是否存在FuncDef
声明  Decl → ConstDecl | VarDecl // 覆盖两种声明
常量声明  ConstDecl → 'const' BType ConstDef { ',' ConstDef } ';' // 1.花括号内重复0次 2.花括号内重复多次
基本类型  BType → 'int' // 存在即可
常数定义  ConstDef → Ident { '[' ConstExp ']' } '=' ConstInitVal  // 包含普通变量、一维数组、二维数组共三种情况
常量初值  ConstInitVal → ConstExp
  | '{' [ ConstInitVal { ',' ConstInitVal } ] '}' // 1.常表达式初值 2.一维数组初值 3.二维数组初值
变量声明  VarDecl → BType VarDef { ',' VarDef } ';' // 1.花括号内重复0次 2.花括号内重复多次
变量定义  VarDef → Ident { '[' ConstExp ']' } // 包含普通变量、一维数组、二维数组定义
  | Ident { '[' ConstExp ']' } '=' InitVal
变量初值  InitVal → Exp | '{' [ InitVal { ',' InitVal } ] '}'// 1.表达式初值 2.一维数组初值 3.二维数组初值
函数定义  FuncDef → FuncType Ident '(' [FuncFParams] ')' Block // 1.无形参 2.有形参
主函数定义   MainFuncDef → 'int' 'main' '(' ')' Block // 存在main函数
函数类型  FuncType → 'void' | 'int' // 覆盖两种类型的函数
函数形参表 FuncFParams → FuncFParam { ',' FuncFParam } // 1.花括号内重复0次 2.花括号内重复多次
函数形参  FuncFParam → BType Ident ['[' ']' { '[' ConstExp ']' }] // 1.普通变量 2.一维数组变量 3.二维数组变量
语句块   Block → '{' { BlockItem } '}' // 1.花括号内重复0次 2.花括号内重复多次
语句块项  BlockItem → Decl | Stmt // 覆盖两种语句块项
语句  Stmt → LVal '=' Exp ';' // 每种类型的语句都要覆盖
  | [Exp] ';'  //有无Exp两种情况
  | Block
  | 'if' '( Cond ')' Stmt [ 'else' Stmt ] // 1.有else 2.无else
  | 'while' '(' Cond ')' Stmt
  | 'break' ';' | 'continue' ';'
  | 'return' [Exp] ';' // 1.有Exp 2.无Exp
  | LVal = 'getint''('')'';'
  | 'printf''('FormatString{,Exp}')'';' // 1.有Exp 2.无Exp
表达式 Exp → AddExp 注: SysY 表达式是int 型表达式 // 存在即可
条件表达式 Cond → LOrExp // 存在即可
左值表达式 LVal → Ident {'[' Exp ']'} //1.普通变量 2.一维数组 3.二维数组
基本表达式 PrimaryExp → '(' Exp ')' | LVal | Number // 三种情况均需覆盖
```

```
数值  Number → IntConst // 存在即可
一元表达式 UnaryExp → PrimaryExp | Ident '(' [FuncRParams] ')' // 3种情况均需覆盖，函数调用也需要覆盖FuncRParams的不同情况
    | UnaryOp UnaryExp // 存在即可
单目运算符 UnaryOp → '+' | '−' | '!' 注：'!'仅出现在条件表达式中 // 三种均需覆盖
函数实参表 FuncRParams → Exp { ',' Exp } // 1.花括号内重复0次 2.花括号内重复多次 3.Exp需要覆盖数组传参和部分数组传参
乘除模表达式  MulExp → UnaryExp | MulExp ('*' | '/' | '%') UnaryExp // 1.UnaryExp 2.* 3./ 4.% 均需覆盖
加减表达式 AddExp → MulExp | AddExp ('+' | '−') MulExp // 1.MulExp 2.+ 需覆盖 3.-需覆盖
关系表达式 RelExp → AddExp | RelExp ('<' | '>' | '<=' | '>=') AddExp // 1.AddExp 2.< 3.> 4.<= 5.>= 均需覆盖
相等性表达式  EqExp → RelExp | EqExp ('==' | '!=') RelExp // 1.RelExp 2.== 3.!= 均需覆盖
逻辑与表达式  LAndExp → EqExp | LAndExp '&&' EqExp // 1.EqExp 2.&& 均需覆盖
逻辑或表达式  LOrExp → LAndExp | LOrExp '||' LAndExp // 1.LAndExp 2.|| 均需覆盖
常量表达式 ConstExp → AddExp 注：使用的Ident 必须是常量 // 存在即可
```

词法分析文档

主要功能

词法分析逐个读取源文件的字符，根据当前字符对类别码进行识别。同时将字符组合成单词进行输出。还要对数字常数完成数字字符到十进制字符的转换。同时删去空格字符和注释。

词法分析实现

采用词法分析单独作为一遍实现，将分析出的类别码放到一个`int SymbolList[100000]`数组中记录，以供语法分析使用。

利用`extern map<int,string> MapSymbol`对类别码和对应编号一一对应，string语句既是最后输出语句。

类别码对应编号

```
#define IDENFR 1
#define INTCON 2
#define STRCON 3
#define MAINTK 4
#define CONSTTK 5
#define INTTK 6
#define BREAKTK 7
#define CONTINUETK 8
#define IFTK 9
#define ELSETK 10
#define NOT 11
#define AND 12
#define OR 13
#define WHILETK 14
#define GETINTTK 15
#define PRINTFTK 16
#define RETURNTK 17
#define PLUS 18
#define MINU 19
#define VOIDTK 20
#define MULT 21
#define DIV 22
```

```
#define MOD 23
#define LSS 24
#define LEQ 25
#define GRE 26
#define GEQ 27
#define EQL 28
#define NEQ 29
#define ASSIGN 30
#define SEMICN 31
#define COMMA 32
#define LPARENT 33
#define RPARENT 34
#define LBRACK 35
#define RBRACK 36
#define LBRACE 37
#define RBRACE 38
#define ERROR 39
#define NOTES 40
```

通过画出文法状态图，知道各类型单词的判断过程。

1. 保留字（字母构成）和标识符（由字母或`_`开始，可以包含数字字母或者`_`）过程

   由字母或者_开始，向后继续查找字母或者数字或者`_`,同时将字符组合起来。通过对`token`字符串和保留字判断，设置对应`symbol`类别码，若不是保留字，即是标识符`IDENFR`。

2. 数字过程

   由数字开始，继续查找数字，整合成数字字符串，转化成数字。

3. 注释过程

   `/* */`注释

   嵌套在判断除号过程中，初始实现思想是当遇到`/*`注释开始，遇到`*/`注释结束。所以需要每次判断当前字符位置/和上一个字符位置*。

   不过遇到这种情况

   ```
   /*/*
       * /testfile created with ``heart'' by 19373384
       *
      /
      */
   ```

   其中/*/也会导致注释结束，所以再次多向前读取一个字符进行判断

   `//`注释

   判断注释到遇到`\n`结束

4. 其他字符

   其他字符直接设置对应的类别码就行。

重要约束

在进入各类型识别函数之前，获得新的字符，在结束识别函数时，对当前字符进行更新。

语法分析文档

语法分析是编译过程的核心部分。语法分析的任务是按照文法，从源程序类别码中识别出个类语法程序，同时进行语法检查，为语义和生成代码做准备。

本次采用递归下降分析法，对文法的每个非终结符都编写对应的子程序，以完成非终结符号所对应的语法成分和识别任务。某个非终结符号的语法分析子程序的功能是，用该非终结符的规则的右部符号串去匹配输入符号串。

整个语法分析从`CompUnit()`函数进入，然后对之后的类别码进行分析。进入一个子程序需要判断FIRST集合，甚至有些语法成分分析的进入，需要预读很多位类别码才能判断。对于BFS中{}一般通过while实现，[]通过if进行实现。

对于具有左递归的文法需要利用相应规则进行改写。

左递归文法的改写

```
AddExp → MulExp | AddExp ('+' | '−') MulExp
-》
AddExp → MulExp { ('+' | '−') MulExp }

RelExp → AddExp | RelExp ('<' | '>' | '<=' | '>=') AddExp
-》
RelExp → RelExp { ('<' | '>' | '<=' | '>=') RelExp }

EqExp → RelExp | EqExp ('==' | '!=') RelExp
-》
EqExp → EqExp { ('==' | '!=') EqExp}

LAndExp → EqExp | LAndExp '&&' EqExp
-》
LAndExp → EqExp { '&&' EqExp }

LOrExp → LAndExp | LOrExp '||' LAndExp
-》
LOrExp → LAndExp { '||' LAndExp }
```

重要约束

虽然有些语法成分不需要输出，但不要擅自省略其对应的子程序函数。否则对整体文法的正确性会有很大影响。

当调用某个分析子程序时，他分析的第一符号`SymbolList[CurSymPos]`；在分析子程序返回报告成功前，将CurSymPos的位置+1，即获得下一个类别码。

错误处理

在程序编写完成后，每一个程序员都无法保证自己的程序完全符合语法、语义规则和其他的一些规定。这些错误如果没有编译器给出的相关提示，想要修改就需要耗费相当多的时间，拖慢编程的效率和准确性。所以编译器对程序的错误处理就显得至关重要。

程序设计要求

根据给定文法设计相关的错误处理程序，能够**将违反常见语法和语义规则的错误反馈给编码者**，同时能进行**错误局部化处理**。反馈方式为当出现错误后，输出相应的错误类别码和错误出现位置所在的行号。不过在本次实验之后，错误处理最好能够详细的说明错误信息，能够帮助完善编译器设计与开发。

对于语法错误，即违反文法定义的编码方式。

对于语义错误，可详见文法定义与相关说明的语义约束部分，还包含超越具体计算机系统的限制。如顶层变量/常量声明，函数定义时不能重复定义同名标识符，此时还需要**符号表管理**来帮助判断是否重名。

对于错误局部化处理，解释是当程序发现错误后，尽可能将错误限制在一个局部的范围内，避免错误扩散和影响程序其它部分的分析和检查。

错误类别码：

用做错误类型的识别，以及输出。

错误类型	错误类别码	解释	对应文法及出错符号(...省略该条规则后续部分)
非法符号	a	格式字符串中出现非法字符 报错行号为**<FormatString>**所在行数。	<FormatString> → ""{<Char>}""
名字重定义	b	函数名或者变量名在**当前作用域**下重复定义。 注意，变量一定是同一级作用域下才会判定出错，不同级作用域下，内层会覆盖外层定义。 报错行号为**<Ident>**所在行数。	<ConstDef>→<Ident> ... <VarDef>→<Ident> ... |<Ident> ... <FuncDef>→<FuncType><Ident> ...
未定义的名字	c	使用了未定义的标识符 报错行号为**<Ident>**所在行数。	<LVal>→<Ident> ... <UnaryExp>→<Ident> ...
函数参数个数不匹配	d	函数调用语句中，参数个数与函数定义中的参数个数不匹配。 报错行号为函数名调用语句的**函数名**所在行数。	<UnaryExp>→<Ident>'('[FuncRParams]')'
函数参数类型不匹配	e	函数调用语句中，参数类型与函数定义中对应位置的参数类型不匹配。 报错行号为函数名调用语句的**函数名**所在行数。	<UnaryExp>→<Ident>'('[FuncRParams]')'
无返回值的函数存在不匹配的return语句	f	报错行号为**'return'**所在行号。	<Stmt>→'return' {'['Exp']'}';'
有返回值的函数缺少return语句	g	只需要考虑函数末尾是否存在return语句即可，**无需考虑数据流**。 报错行号为函数**结尾的'}'**所在行号。	FuncDef → FuncType Ident '('[FuncFParams] ')' Block
不能改变常量的值	h	<LVal>为常量时，不能对其修改。 报错行号为**<LVal>**所在行号。	<Stmt>→<LVal>'=' <Exp>';'|<LVal>'=' 'getint' '(' ')' ';'
缺少分号	i	报错行号为分号**前一个非终结符**所在行号。	<Stmt>,<ConstDecl>及<VarDecl>中的';'
缺少右小括号')'	j	报错行号为右小括号**前一个非终结符**所在行号。	函数调用(<UnaryExp>)、函数定义(<FuncDef>)及<Stmt>中的')'
缺少右中括号']'	k	报错行号为右中括号**前一个非终结符**所在行号。	数组定义(<ConstDef>,<VarDef>,<FuncFParam>)和使用(<LVal>)中的']'
printf中格式字符与表达式个数不匹配	l	报错行号为**'printf'**所在行号。	Stmt →'printf''('FormatString{,Exp}')'';'
在非循环块中使用break和continue语句	m	报错行号为**'break'**与**'continue'**所在行号。	<Stmt>→'break'';' |'continue'';'

符号表管理

符号表适用于保存每个标识符及其属性信息的数据结构，在符号表的表型中登录标识符的属性。使用符号表能快速地找到每个标识符的表项并能快速存储和检索表项记录的数据信息。

在多变扫描的编译程序中，在词法分析阶段就将标识符填入符号表，标识符的其他属性则在语义分析和代码生成阶段填入。

符号表的构建方式

符号表的存取操作可能会花去编译期间很多的处理器时间，所以需要合理的选择符号表的组织形式，选择高效的填表查表方式，这样才能提高编译程序的工作效率。

符号表典型结构

token/Name	标识符类型	数据类型	行号
	函数	int	
	常量	array1	
	变量	array2	
	形参	void	

对于一些特殊的类型需要考虑更多的信息：

1. **数组**

   维数、上下界值

2. **函数**

   形参个数，形参类型，**实参个数**，实参类型，所在层次，函数返回值。

其中**输出行号**，可以通过 `SymbolListLine` 对应位置获得。不需要通过符号表获得。但也可以将对应行号存入 `SymbolTable` 结构中

符号表组织形式

采用栈式符号表进行组织，同时利用分程序索引表保存程序层次。

符号表典型操作

1. **填表**

   找到与标识符有关的文法，对标识符的信息进行分析，将新纪录推入栈顶单元即可。要考虑标识符在同一分程序同名情况。

   标识符Name，存放在TokenList string数组

2. **查表**

   需要将符号表填表和查表操作进一步独立

   什么情况下查表：

   声明部分时：查询该标识符在所在程序单元 `从栈式符号表栈顶到分程序索引表栈顶对应的栈式符号表位置` 中是否存在同名的标识符，查表函数的形参为 `token字符串` ，函数类型 `int` 返回 `同名标识符在栈式符号表的下标` 或者 `-1` 。

   调用部分：查询所在程序单元的符号表中是否存在同名的标识符，若存在，说明该标识符已声明；若不存在，继续向外层程序单元进行查询，直至到最外层单元仍未出现同名标识符时，说明该标识符未声明。

   分程序的开始：执行定位操作

   分程序的结束：执行重定位操作

3. **定位**

   分程序索引表的顶端填入新的分程序索引项。（即分程序头一个标识符在符号表的位置）

   定位时机在判断当前标识符是一个函数名时

> 不能只考虑函数的定位和重定位，在while if else 等含有block的语句都能声明新的变量。可是while if else 都没有变量名 如何记录变量名在栈中的位置，while if else 都是语句在stmt中的block 前进行定位，在block后进行重定位，即可

4. **重定位**

   清除刚刚被编译完的分程序在栈式符号表中的所有记录。

   考虑分程序符号表为空情况

   > //重定位时机在block `}`结束后,**错误**
   >
   > 应该是{},之间一一对每次大括号都要定位，但是
   >
   > 在函数结束时重定位，应该是funcdef }
   >
   > 对于在main函数之前的函数的重定位情况

哪些文法需要填表

```
/*声明*/
ConstDef → Ident { '[' ConstExp ']' } '=' ConstInitVal
VarDef → Ident { '[' ConstExp ']' }| Ident { '[' ConstExp ']' } '=' InitVal
FuncDef → FuncType Ident '(' [FuncFParams] ')' Block
/*调用*/
LVal → Ident {'[' Exp ']'}
UnaryExp → PrimaryExp | Ident '(' [FuncRParams] ')'
```

> **疑惑**
>
> - 形参是多维数组时， int [] [`Constexp`] 的`Constexp`会不会出现ident 。

错误处理实现

在用递归下降分析法进行语法分析是，一旦检查出错误就将错误性质送给变量cx（行号，错误类型），然后转出错处理,根据cx输出错误的类型和对应的行号，并且还能保证能够运行之后的程序。

对以上错误类型进行处理。

首先将错误分类成语法错误和语义错误。语法错误包含以下几项

语法错误

缺少分号	i	报错行号为分号**前一个非终结符**所在行号。	<Stmt>,<ConstDecl>及<VarDecl>中的';'
缺少右小括号')'	j	报错行号为右小括号**前一个非终结符**所在行号。	函数调用(<UnaryExp>)、函数定义(<FuncDef>)及<Stmt>中的')'
缺少右中括号']'	k	报错行号为右中括号**前一个非终结符**所在行号。	数组定义(<ConstDef>,<VarDef>,<FuncFParam>)和使用(<LVal>)中的']'

缺少分号

同下

缺少右小括号

不仅仅要输出出错的位置，还要对错误进行改正，防止影响之后的程序错误判断。比如if()缺少，会导致跳出if else整体判断，if之后若紧跟else，那么无法对else进行分析，导致死循环。要进行错误修正。修正后返回语句的开始进行判断。

在返回语句开始的同时需要注意语句中是否会有函数的声明，即是否会造成标识符入符号表栈，如果有，需要将标识符弹出。

突然想到好像不用错误修正，直接在error后继续本应该的判断就行。

缺少右中括号

实现以上三中缺少对应右部匹配符号的错误处理，当前`SymbolList[CurSymPos]`类别码已经找不到正确的引号小括号中括号。

非法符号实现

出现位置，在`printf`语句的字符串中。`ForMatString`字符串在词法分析中就已经被放入token中。可以在语法分析`ForMatString`函数中新建一个局部变量`ForMatStringFlag`局部变量对当前token进行判断，是否存在非法符号。

非法符号定义为：32，33，40-126之外的ASCII字符，若出现\则后边必须跟n

报错行号为`STRON`类别码对应的行号，即`SymbolListLine[CurSymPos]`

错误处理结束后，进行下一个字符的判断（当前是`strcon`如果不读下一个，那么跳出`formatstring`函数后，之后的递归下降判断无法通过，会导致死循环）

语义错误

名字重定义

名字重定义	b	函数名或者变量名在**当前作用域**下重复定义。 注意，变量一定是同一级作用域下才会判定出错，不同级作用域下，内层会覆盖外层定义。 报错行号为**<Ident>**所在行数。	<ConstDef>→<Ident> ... <VarDef>→<Ident> ... |<Ident> ... <FuncDef>→<FuncType><Ident> ...

出现在变量常量函数函数参数的声明中，只需在插入符号函数中进行判断即可

> ？？？名字重定义后，不填入符号表，输出行号，也就是当前类别码位置地行号，还要进行什么操作，

未定义的名字

未定义的名字	c	使用了未定义的标识符 报错行号为**<Ident>**所在行数。	<LVal>→<Ident> ... <UnaryExp>→<Ident> ...

调用部分，符号表中是否有过声明，即是栈式符号表中是否有对应的标识符。

函数参数个数不匹配

函数参数个数不匹配	d	函数调用语句中，参数个数与函数定义中的参数个数不匹配。 报错行号为函数名调用语句的**函数名**所在行数。	<UnaryExp>→<Ident>'('[FuncRParams]')'

需要对符号表进行增加，记录函数参数的个数，在调用的时候进行判断。新建一个结构体保存函数参数的个数和类型。在函数参数声明 和函数参数调用时进行调用。

声明时，在进入形参函数之前定义一个`PramNum`，将`int &PramNum`作为形参函数的形参，用来返回改变后的`ParmNum`值。

`UnaryExp` 调用，首先不会知道标识符的数据类型，只有在栈中找到标识符的声明之后，获得标识符的数据类型。如果是函数标识符类型，获得其函数的形参个数 `n` 和声明标识符在符号表中的位置，将这些作为实参表的形参，在实参表函数中进行判断。对其后的n个标识符 `即形参` 和当前标识符之后的的 `n`。对而标识符函数调用时，函数参数和函数个数在完成函数实参表时才能够获取到。

函数参数类型不匹配

函数参数类型不匹配	e	函数调用语句中，参数类型与函数定义中对应位置的参数类型不匹配。 报错行号为函数名调用语句的**函数名**所在行数。	<UnaryExp>→<Ident>'('[FuncRParams]')'

不应该是对函数名之后n个变量进行判断，符号表不能这样存储形参，因为在定义的函数结束后，除了函数名保存在栈符号表中外，其他局部变量全部弹出。在之后调用函数时，其形参已经不在符号表中。在其他信息结构体中添加新的数组，用来存储各个形参类型。在函数名对应位置的补充信息中的形参表收集从FuncFParams 获得的

令FuncFParams 改变ExtraSymbol，令FuncFParam返回DataType。

不能改变常量的值

从Lval返回变量类型，对之后的等号进行判断